A challenge for atomicity verification

Hesselink, Wim H.

Published in:
Science of computer programming

DOI:
10.1016/j.scico.2008.01.001

Publication date:
2008

Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 71 (2008) 57–72
Science of Computer Programming
www.elsevier.com/locate/scico

A challenge for atomicity verification

Wim H. Hesselink*

Department of Mathematics and Computing Science, University of Groningen, P.O. Box 407, 9700 AK Groningen, The Netherlands

Received 8 January 2007; received in revised form 24 December 2007; accepted 2 January 2008
Available online 6 January 2008

Abstract

An unpublished algorithm of Haldar and Vidyasankar implements an atomic variable of an arbitrary type T for one writer and one reader by means of 4 unsafe variables of type T, three two-valued safe variables, and one three-valued regular variable. We present this algorithm, and prove its correctness by means of a refinement towards a known specification of an atomic variable. The refinement is a composition of refinement functions and a forward simulation. The correctness proof requires many nontrivial invariants. In its construction, we relied on the proof assistant PVS for the administration of invariants and proofs and the preservation of consistency.

Keywords: Atomicity; Safe variables; Regular variables; Refinement; Theorem proving

1. Introduction

The design of an algorithm ideally starts with the specification and proceeds via a number of motivated refinement steps to conclude with an implementation. In the field of concurrency, however, this ideal is more often proclaimed than executed. In our practice, more often, the designer has an idea for a concurrent algorithm, and while trying to develop and prove the algorithm, he changes the algorithm until it is concrete enough and has an associated proof. Just as often, at the other side of the spectrum, there may be an algorithm with or without any proof, and we may want to decide its correctness (and the reliability of the proof).

In any case, we want our methods for the analysis of algorithms to be strong enough to ascertain or refute their correctness. Naturally, most publications with a methodological aim present idealized design scenarios and algorithms that fit such an idealized design, but most algorithms are not designed for methodological purposes, and can yet be presented with a more or less fitting proof.

In the present case, we want to present an algorithm, not because it has important applications, but mainly because it offers a challenge to the research community and a little bit because it has an instructive proof. We present it with a refinement proof that has been constructed and verified with the proof assistant PVS of [10]. The challenge is to present the algorithm with a more natural design and a more intuitive verification. For now, however, we have no alternative to offer.

* Tel.: +31 503633933; fax: +31 503633800.
E-mail address: w.h.hesselink@rug.nl.

0167-6423/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.01.001

In the nineties, Haldar, Subramanian, and Vidyasankar set themselves the task to construct an atomic variable of some arbitrary type *Item* for one writer and one reader by means of unsafe variables of type *Item* together with some nonatomic control variables, preferably boolean. The paper of [3] presents a construction C1 that uses four unsafe variables of type *Item* together with three safe boolean variables, in which the writer may have to do two *Item* assignments. We gave an assertional proof of algorithm C1 in [5].

The algorithm we treat here is construction C2 of Haldar and Vidyasankar (private communication, 2006) of an atomic variable of type *Item* for one reader and one writer in which the writer does only one *Item* assignment. It again uses four unsafe variables of type *Item*, but now three safe boolean variables and one regular three-valued variable (the latter can be replaced by two safe boolean variables). Algorithm C2 is more difficult than C1 because the writer only writes once and subtle communication with the reader is needed to ensure that it writes under mutual exclusion and is yet at a place where the reader can read next time.

Haldar and Vidyasankar have conjectured (unpublished) that the algorithms C1 and C2 are optimal in the sense of using minimal sets of nonatomic variables and assignments to them. So, if it can be argued that algorithm C2 is ugly, it is not just a random algorithm.

The correctness issue of an algorithm that implements atomic variables by means of nonatomic ones always leads to two questions. How to model the actions on the nonatomic variables? And how to decide that the algorithm indeed implements an atomic variable?

With respect to the first point, we present a novel treatment of unsafe, safe, and regular variables. We model them by extending the assignments to them with alternative temporary assignments. We then use a refinement function to eliminate the extra variables introduced, at the cost of making inspection of the nonatomic variables more nondeterministic.

As for the second point, in [4], we determined a criterion for atomicity that was based on the definition of atomicity in [9], and we used this to prove atomicity for the algorithms of [2,11]. Afterwards, we felt that the definition of atomicity in [9] was too complicated and verbose for such an elementary property as atomicity. We therefore developed and proved a related refinement criterion for atomicity in [6]. This criterion is simplified here to a specification *HqRW* of an atomic read–write variable to which it is suitable to make refinement functions.

The criterion is applied as follows. After the elimination of the extra variables mentioned above, we use a forward simulation to introduce ghost variables to construct a refinement function towards *HqRW*. This gives us six proof obligations for atomicity. These proof obligations are just safety conditions, i.e., invariants. Four of them are easily resolved. The two remaining proof obligations turn out to be challenging. They require more than twenty auxiliary invariants. It is here that, in an earlier effort, we needed a model checker to support our failing intuition for reliable invariants. In this earlier effort, we remained in a single state space and did not use refinement functions and forward simulations.

The present treatment with refinement functions and forward simulations gives a cleaner separation between modelling, proof obligations, and auxiliary constructions. Yet in the end, the number of invariants required to fulfill the proof obligations is roughly the same. The new approach has the additional advantage that it can easily include a minor variation of algorithm C2 in which the regular three-valued variable is replaced by two safe boolean variables.

Overview. In Section 2, we discuss and formalize the various forms of nonatomic shared variables that occur in the algorithm. Algorithm C2 itself is presented in Section 3. In Section 4, we present the specifications *ARW* and *HqRW* of atomic read–write variables and describe the concepts of implementation and simulation used to relate different specifications, following [1,6].

In Section 5, algorithm C2 is transformed into a specification *HVc2* in the same format as *HqRW*. Subsequently, we eliminate the nonatomic variables via a refinement function towards a specification *HVc2a*, and we then prove that the unsafe variables are treated under mutual exclusion as required.

In Section 6, we construct a forward simulation towards a specification *HVc2b* with "ghost variables" that are needed for a refinement function towards *HqRW*. The requirements for refinement functions then induce six proof obligations. These proof obligations are resolved by introducing and proving a host of invariants.

Section 7 contains the variation (*HVac2*) of the algorithm in which the three-valued regular variable is replaced by two safe bit variables. In Section 8 we draw the conclusions.

The PVS proof of algorithm C2 consists of the construction and verification of a simulation relation from specification *HVc2*, via *HVc2a* and *HVc2b*, towards *HqRW*. It also contains a refinement function from the

alternative specification *HVac2* towards *HVc2a*. This proof is available at www.cs.rug.nl/~wim/mechver/atomicity/index.html.

2. Nonatomic shared variables

We first have to recall or introduce the concepts of safeness, regularity, and atomicity of shared variables. In this section, we allow arbitrary many processes, but for simplicity we only consider shared variables that are written by a single process. Since the processes themselves are sequential, it follows that different write operations to the same variable never overlap.

The minimal correctness assumption for a shared variable is therefore that every read operation that does not overlap with any write operation returns the most recently written value. Following [7], a shared variable is called *safe* iff every read operation that overlaps with a write operation returns some legitimate value of the domain of the variable. A variable is called *unsafe* iff it is not safe. For an unsafe variable, a read operation is not allowed to overlap with write operations since otherwise chaos may result.

A shared variable x is called *regular* iff a read operation on x concurrent with some write operations may return either the old value of x, or any of the values that are being written concurrently. Clearly, every regular variable is safe, but if the type of x has three or more values, a safe variable x need not be regular. Since we can ensure that a writer only writes $\mathtt{x} := E$ under the precondition $\mathtt{x} \neq E$, an algorithm with regular boolean variables can be easily transformed into one with safe boolean variables.

Finally, a shared variable is *atomic* iff read and write operations behave as if they never overlap but always occur in some total order that refines the *precedence* order (an operation *precedes* another iff it terminates before the other starts).

We introduced a formal model for safe variables in [5]. We now extend this model with regular and unsafe variables. Indeed, since assertional reasoning depends on the states between the atomic steps, we need to model the actions as atomic steps of some kind. By convention, shared variables are written in typewriter font and private variables are written slanted.

A read action of a shared variable x into a private variable v is written $v := \mathtt{x}$. It can be regarded as atomic, since it does not influence the shared state. We only need to reckon with the possibility that it overlaps with one or more write actions to x.

For a safe shared variable, we need to indicate that reading during a write action can return any value. We therefore denote a write action of a private expression E into a safe shared variable x by

$$\text{(flickering) } \mathtt{x} := E. \tag{1}$$

In relational semantics like TLA of [8], we model this by a predicate like

$$pc' = pc \quad \vee \quad (pc' = pc + 1 \ \wedge \ \mathtt{x}' = E). \tag{1a}$$

The primes are used for the values of the variables after the step, *pc* stands for the location pointer, and by convention all variables apart from *pc* and x are unchanged. In other words, command (1) is modelled in (1a) as a repetition of arbitrary assignments to x that ends with the actual assignment of E to x. The value of x during the repetition is indeterminate. We assume the liveness condition that the repetition terminates.

For a regular variable, reading during concurrent write actions can return either the old value or any of the new values. So, there is the same kind of flickering, but only between the old and the new values. We denote a write action of a private expression E into a regular shared variable x by

$$\text{(regular) } \mathtt{x} := E. \tag{2}$$

In order to model this, we introduce a second variable xS (to be called the *shadow* of x) of the same type as x to hold the latest correct value of x. The step then gets the relational meaning given by

$$\begin{array}{l}(pc' = pc \ \wedge \ \mathtt{xS}' = \mathtt{xS} \ \wedge \ (\mathtt{x}' = \mathtt{xS} \ \vee \ \mathtt{x}' = E)) \\ \vee \quad (pc' = pc + 1 \ \wedge \ \mathtt{x}' = \mathtt{xS}' = E).\end{array} \tag{2a}$$

Indeed, during a write action, any reading process can read the old or the new value. When a read action overlaps with several write actions, the result can be the old value or any of the new values, since conceptually the read action is shrunk to an atomic one that coincides with the write action of the value that turns out to have been read.

A write action of a private expression E into an unsafe shared variable $\mathtt{x}$ is written

$$(\text{unsafe})\ \mathtt{x} := E. \tag{3}$$

Inspired by a suggestion of Lamport, this is formalized as a two-step write operation. If $\mathtt{x}$ is of type T, we allow $\mathtt{x}$ temporarily to hold values of type $T \cup \{\bot\}$, and we assume that reading $\bot$ into a private variable leads to chaotic behaviour. The step gets the relational meaning of first repeatedly writing $\bot$, concluded by the actual assignment:

$$(pc' = pc \ \wedge\ \mathtt{x}' = \bot) \quad \vee \quad (pc' = pc + 1 \ \wedge\ \mathtt{x}' = E). \tag{3a}$$

This leads to the proof obligation $v \neq \bot$ for all private variables v used for storing the result of reading $\mathtt{x}$. Of course, we have the additional complications that $\mathtt{x} \neq \bot$ holds initially, and that expression E is of type T, i.e. $\neq \bot$.

Since there is only one process writing $\mathtt{x}$, we do allow expression E in (1), (2), or (3) to contain $\mathtt{x}$. Then E is not private and, in order to transform (1) into (1a), etc., we use the shadow $\mathtt{xS}$ as a private variable of the writer to hold the old value of $\mathtt{x}$, so that expression E is truly private and the transformation is allowed. For this purpose, we extend (1a) and (3a) with conjuncts expressing that $\mathtt{xS}$ is modified only in the concluding assignment, just as in (2a). In all three cases, we have the obvious invariant that $\mathtt{xS} = \mathtt{x}$ unless the unique process that can assign $\mathtt{x}$ is indeed assigning $\mathtt{x}$.

The above assertional characterizations of safe, regular, and unsafe variables are nontrivial consequences of the behavioural definitions, also valid when there is more than one reading process. The translation from the behavioural to the assertional setting requires rather delicate behavioural arguments. Since such a translation would distract us from our central aims, we just postulate the above characterizations.

3. Algorithm C2

Algorithm C2 of Haldar and Vidyasankar implements an atomic variable of type *Item* written by one writer process and read by one reader process by means of four unsafe variables of type *Item*, three safe variables of a two-valued domain for the writer, and one regular three-valued variable for the reader. The algorithm uses the following shared variables:

type $Bit = \{0, 1\}$, $Tri = \{0, 1, 2\}$;
var $\mathtt{buf}$: **array** Bit, Bit **of** *unsafe Item* ;
$\mathtt{ww}, \mathtt{cc}$: *safe Bit* ; $\mathtt{fw}$: *safe Bool* ; $\mathtt{rr}$: *regular Tri* .

The basic idea is that, roughly speaking, $\mathtt{buf}[\mathtt{ww}, \mathtt{cc}]$ holds the latest item written. If the reader does not read, the writer toggles $\mathtt{cc}$ at every write action. When the reader reads a new item, it sets $\mathtt{rr} := \mathtt{ww}$. When the writer notices this after writing, it sets $\mathtt{fw}$ to true to notify the reader that it will toggle $\mathtt{ww}$ at its next write action.

The reader only reads $\mathtt{buf}[\mathtt{rr}, lc]$, where $\mathtt{rr}$ and lc are intended as copies of $\mathtt{ww}$ and $\mathtt{cc}$, respectively. To remember the latest item in cases where actual reading is not necessary, the reader has a persistent private variable *result*:

privar *result* : *Item* .

The initial state is required to satisfy

$$(\neg \mathtt{fw} \ \vee\ \mathtt{ww} = \mathtt{rr}) \ \wedge\ (\forall\, i, j : \mathtt{buf}[i, j] = item0) \ \wedge\ result = item0 \ .$$

Writing an argument *arg* is implemented by

proc *write*(*arg*) :
  if $\mathtt{fw}$ **then**
    (unsafe) $\mathtt{buf}[1 - \mathtt{ww}, \mathtt{cc}] := arg$;
    (flickering) $\mathtt{ww} := 1 - \mathtt{ww}$;
    (flickering) $\mathtt{fw} := false$;
  else

```
        (unsafe) buf[ww, 1 − cc] := arg ;
        (flickering) cc := 1 − cc ;
        if rr = ww then
            (flickering) fw := true end ;
    end
end write .
```

The reader first tests fw twice, sets *lc* to cc, and reads buf[rr, *lc*]. If one of the tests fails, it sets rr to a local copy of ww, and proceeds as before:

```
proc read() :
    var lw, lc : Bit ;
    if fw then
        lc := cc ;
        if fw then
            result := buf[rr, lc] ; return ;
        end
    end ;
    lw := ww ;
    if lw ≠ rr then
(*)     (regular) rr := 2 ;
        (regular) rr := lw ;
        lc := cc ;
        result := buf[rr, lc] ;
    end
end read .
```

The reader's assignment (*) rr := 2 is not superfluous. If we would omit it, the value of rr would flicker between 0 and 1 during the next assignment. Therefore, the writer could infer almost nothing from reading rr and, hence, could not avoid writing an element of buf while the reader would be reading there. The following scenario for the case that (*) is omitted shows what can go wrong. Let the reader start reading in a state with ¬fw and rr = cc = 0 and ww = 1. The reader sees ¬fw, sets *lw* := 1, and starts toggling rr between 0 and 1. During this toggling, the writer writes four times: firstly, it writes buf[1, 1], sets cc := 1, reads rr = 1 = ww and sets fw := *true*; secondly it writes buf[0, 1] and sets ww := 0, fw := *false*; thirdly it writes buf[0, 0], sets cc := 0, reads rr = 0 = ww and sets fw := *true*; fourthly it writes buf[1, 0]. During the fourth writing, the reader terminates its toggling, sets rr := 1 and *lc* := 0, and reads buf[1, 0]. This leads to chaotic results because the writer is writing there. Assignment (*) in algorithm C2 precludes this scenario because rr can now toggle only between 0 and 2, or 2 and 1.

Unfortunately, we cannot explain why the algorithm with (*) included should be correct. Indeed, we started the verification with an open mind to prove or to refute the correctness, and we now only accept the correctness because of our mechanically verified proof.

Notice that procedure *read* does not actually read in a situation where it finds fw false and ww = rr. In that case, the reader just returns the old value of *result*. This may hold, e.g., in the initial state when the writer has not yet written.

4. Specifications of atomicity

We use the modelling of atomic variables of [6], but we specialize to the case of a single reader and a single writer. The procedure calls are modelled with potentially infinite loops. For both reader and writer we reserve location 0 for the environment that may call them and that will provide the writer with its arguments. The environment is not subject to liveness conditions. The writer and reader use locations from 20 and 50 onward, and are subject to the fairness condition that they always eventually reestablish *pc* = 0.

The environment only acts when *pc* = 0. We write *pw* and *pr* for *pc* of writer and reader, respectively. We give the environment a variable out to observe the *result* of reading, because the moment this *result* becomes available must not be observable.

Environment:

$$[]\ pw = 0 \ \rightarrow\ arg := \mathit{arbitrary}\ ;\ pw := 20\ .$$
$$[]\ pr = 0 \ \rightarrow\ pr := 50\ .$$
$$[]\ pr = 0 \ \rightarrow\ \mathtt{out} := \mathit{result}\ .$$

The abstract specification ARW of a read–write system with two processes just uses a single shared variable, say

var reg : *Item* ;

and the processes are modelled by:

Writer:
20: reg := *arg* ; **goto** 0 .
Reader:
50: *result* := reg ; **goto** 0 .

4.1. Implementation as observable trace inclusion

What does it mean that algorithm C2 implements specification ARW? For this, we use the theory of [1] in the version of [6]. According to this theory, an algorithm *implements* a specification if every behaviour of the algorithm is observably equivalent to a behaviour of the specification. Here, a *behaviour* is an infinite sequence of subsequent states that starts in an initial state, does steps of the algorithm or specification, and satisfies the liveness conditions imposed. Two infinite sequences of states are *observably equivalent* if both sequences can be extended with stuttering steps to sequences such that, for every index, the pair of states at that index is observably equivalent. Two states are *observably equivalent* if their observable variables are equal.

The standard methods to prove implementation relations are by means of refinement functions and forward simulations. For simplicity of presentation, we here ignore the liveness conditions for all algorithms and specifications. We refer to [1,6] for details.

If K and L are specifications, a function $f : \mathit{states}(K) \rightarrow \mathit{states}(L)$ is a *refinement function* from K to L if, for every initial state x of K, the state $f(x)$ is initial for L, and there is an invariant J of specification K such that for every step (x, x') of K with $x \in J$ the pair $(f(x), f(x'))$ is a step of L.

A binary relation R between $\mathit{states}(K)$ and $\mathit{states}(L)$ is a *forward simulation* from K to L if, for every initial state x of K, there is an initial state y of L with $(x, y) \in R$, and there is an invariant J of K such that, for every step (x, x') of K and every state y of L with $(x, y) \in R$ and $x \in J$, there is a step (y, y') of L with $(x', y') \in R$.

4.2. The specification HqRW

We apply the theory of [6] to prove that the algorithm C2 implements specification ARW. For this purpose, we developed a more liberal abstract specification $HqRW$ of an atomic variable. It uses the declarations:

var hist : $\mathbb{N} \rightarrow \mathit{Item} \cup \{\bot\}$;
var masq : $\mathbb{N}$;
privar *firstW* , *sqnW* , *firstR* , *sqnR* : $\mathbb{N}$;
initially $\mathtt{hist}(\mathtt{masq}) = \mathit{item0} \ \wedge\ (\forall\, n : n \neq \mathtt{masq} \ \Rightarrow\ \mathtt{hist}(n) = \bot)$.

The writer and reader have private variables *firstW* , *sqnW* and *firstR* , *sqnR* , respectively. Function hist stands for the history of written values, *sqnW* and *sqnR* stand for sequence numbers, masq stands for the maximum sequence number, and *firstW* and *firstR* stand for the value of masq when the process starts its current operation.

The writer and reader are specified by

Writer:
20: *firstW* := masq ; **goto** 21 .
21: **choose** *sqnW* **with** $\mathit{firstW} < \mathit{sqnW} \ \wedge\ \mathtt{hist}(\mathit{sqnW}) = \bot$;
hist(*sqnW*) := *arg* ; **goto** 22 .
22: masq := max(masq, *sqnW*) ; **goto** 0 .

So the writer puts its argument somewhere in the sequence `hist`.

Reader:

```
50:     firstR := masq ; goto 51 .
51:     choose sqnR with firstR ≤ sqnR ∧ hist(sqnR) ≠ ⊥ .
        result := hist(sqnR) ; goto 52 ;
52:     masq := max(masq, sqnR) ; goto 0 .
```

The reader obtains its *result* from the sequence `hist`. The numbers `masq`, *sqnW*, and *sqnR* serve to ensure linearizable behaviour.

In [6], it is proved that a minor variation of this specification called *HRW* implements the abstract specification *ARW*. Actually, in *HRW* the commands 22 and 52 are split over two locations, but since the values of *pc* are private and not observable, the quicker version *HqRW* implements the slightly slower version *HRW*. This is proved in the new PVS proof of [6], by means of a so-called gliding simulation. We thus have that *HqRW* implements *ARW*.

5. Specification *HVc2*

The question is now whether algorithm C2 implements specification *HqRW*. We therefore reformulate algorithm C2 as a specification *HVc2* in the format of specifications *ARW* and *HqRW*. In accordance with Section 2, we introduce shadow variables `wwS`, `ccS`, and `rrS` with the same types and initializations as `ww`, `cc`, and `rr`, respectively. The shadow of `fw` is unnecessary because `fw` is safe and is not read by the writer. So, the state space of *HVc2* is spanned by the variables mentioned in Section 3 together with the shadow variables `wwS`, `ccS`, `rrS`, and the program counters *pw* and *pr*.

The writer is transformed into:

Writer:

```
20:     goto (fw ? 21 : 24) .
21:     buf[1 − ww, cc] := arg ; goto 22 ;
    []  buf[1 − ww, cc] := ⊥ ; goto 21 .
22:     ww := wwS := 1 − wwS ; goto 23 ;
    []  ww := arbitrary ; goto 22 .
23:     fw := false ; goto 28 ;
    []  fw := arbitrary ; goto 23 .
24:     buf[ww, 1 − cc] := arg ; goto 25 ;
    []  buf[ww, 1 − cc] := ⊥ ; goto 24 .
25:     cc := ccS := 1 − ccS ; goto 26 ;
    []  cc := arbitrary ; goto 25 .
26:     goto (rr = ww ? 27 : 28) .
27:     fw := true ; goto 28 ;
    []  fw := arbitrary ; goto 27 .
28:     goto 0 .
```

The writer has nondeterministic choices in 21, 22, 23, 24, 25, 27 corresponding to the unsafe assignments to `buf` and the flickering assignments to `ww`, `fw`, and `cc`. Notice that, in each of these cases, the second alternative does not change the location, since it is an incomplete assignment.

Reader:

```
50:     goto (fw ? 51 : 53) .
51:     lc := cc ; goto 52 .
52:     goto (fw ? 57 : 53) .
53:     lw := ww ; goto (lw = rr ? 58 : 54) ;
54:     rr := rrS := 2 ; goto 55 ;
    []  rr := (2 or rrS) ; goto 54 .
```

55: rr := rrS := *lw* ; **goto** 56 ;
[] rr := (*lw* **or** rrS) ; **goto** 55 .
56: *lc* := cc ; **goto** 57 .
57: *result* := buf[rr, *lc*] ; **goto** 58 .
58: **goto** 0 .

In 54 and 55, the reader may repeatedly perform incomplete assignments to the regular variable rr.

In this way, the algorithm is turned into a specification *HVc2*. Its state consists of the values of all variables, including the location counters *pw* of the writer and *pr* of the reader. In every state, either process is able to do one or more steps. For example, in a state with $pw = 23$ and $pr = 52$ and $\mathtt{fw} = \mathit{false}$, the writer can make $\mathtt{fw} := \mathit{true}$ or $pw := 28$, or the reader can make $pr := 53$.

Specification *HVc2* gives the proof obligation that $\mathtt{rr} \neq 2$ at 57. This proof obligation will be resolved by invariant *Iq5* below. The proof obligation for the unsafe variables buf will be resolved in Section 5.3.

We then proceed to prove that specification *HVc2* implements *HqRW*. This is done by means of three implementation relations, viz. a refinement function from *HVc2* to *HVc2a*, a forward simulation from *HVc2a* to *HVc2b*, and a refinement function from *HVc2b* to *HqRW*.

5.1. Invariants for HVc2

In order to eliminate the shadow variables, we develop some easy invariants. During the design, the list of invariants continuously changes: new ones are proposed, some invariants turn out to be invalid, some invariants are abolished as superfluous since they are implied by other ones. Also, the order of presentation changes. The list lives in two different documents: the document of the conceptual proof and the PVS proof script. For the ease of finding and replacing the invariants consistently, without replacing other identifiers by mishap, we give them systematic names of the form *Xqd* where *X* gives an indication of the kind of invariant, the *q* is to preclude mishap, and the *d* is a sequence number < 10.

It is clear that ww and cc are equal to their shadows at all locations without assignments to them:

Iq0: $\mathtt{ww} = \mathtt{ssS} \quad \vee \quad pw = 22$,
Iq1: $\mathtt{cc} = \mathtt{ccS} \quad \vee \quad pw = 25$.

The test of fw in 20 entails the obvious invariants:

Iq2: $pw \in \{21, 22\} \quad \Rightarrow \quad \mathtt{fw}$,
Iq3: $pw \in \{24, 25, 26\} \quad \Rightarrow \quad \neg\mathtt{fw}$.

For the reader, we observe that the regular variable rr always equals its shadow or, at an assignment location, possibly, the new value:

Iq4: $\mathtt{rr} = \mathtt{rrS} \quad \vee \quad (pr = 54 \ \wedge \ \mathtt{rr} = 2) \quad \vee \quad (pr = 55 \ \wedge \ \mathtt{rr} = lw)$.

The variables rr and rrS get their values from *lw*. We therefore have:

Iq5: $\mathtt{rr} \neq 2 \quad \vee \quad pr \in \{54, 55\}$,
Iq6: $pr = 54 \quad \Rightarrow \quad \mathtt{rrS} = 1 - lw$,
Iq7: $pr = 55 \quad \Rightarrow \quad \mathtt{rrS} = 2$.

In the proof of *Iq6*, we use that *Iq4* and *Iq5* imply that $\mathtt{rr} = \mathtt{rrS} \in \{0, 1\}$ at 53, and that the test in 53 gives $\mathtt{rr} \neq lw$.

5.2. A more abstract algorithm

We now simplify specification *HVc2* to *HVc2a*, where the variables wwS, ccS, *lw*, and rrS are eliminated. The state space of *HVc2a* is spanned by the variables

buf : **array** *Bit*, *Bit* **of** *Item* ;
ww, cc, rr, *lc* : *Bit* ; fw : *Bool* ; *pw*, *pr* : $\mathbb{N}$.

We move the assignments in 22, 25, 27 one location upward, and make them atomic. The nondeterminacy entailed by the nonatomic assignments is made explicit at the locations where these variables are being read.

Writer of HVc2a:

20: **goto** (fw ? 21 : 24) .
21: buf[1 − ww, cc] := *arg* ; ww := 1 − ww ; **goto** 22 ;
[] buf[1 − ww, cc] := ⊥ ; **goto** 21 .
22: **goto** 23 .
23: fw := *false* ; **goto** 28 .
24: buf[ww, 1 − cc] := *arg* ; cc := 1 − cc ; **goto** 25 ;
[] buf[ww, 1 − cc] := ⊥ ; **goto** 24 .
25: **goto** 26 .
26: ww = rr ≢ *pr* = 54 → fw := *true* ; **goto** 27 ;
[] ww = rr ∨ *pr* ∈ {54, 55} → **goto** 28 .
27: **goto** 28 .
28: **goto** 0 .

The nondeterministic choice at 26 when $pr \in \{54, 55\}$ reflects the flickering status of rr in 54 and 55. Note that the operator ≢ at 26 is an "exclusive or".

Reader of HVc2a:

50: fw → **goto** 51 ;
[] ¬fw ∨ *pw* ∈ {23, 27} → **goto** 53 .
51: *lc* := cc ; **goto** 52 ;
[] *pw* = 25 → *lc* := 1 − cc ; **goto** 52 .
52: fw → **goto** 57 ;
[] ¬fw ∨ *pw* ∈ {23, 27} → **goto** 53 .
53: rr = ww ∨ *pw* = 22 → **goto** 58 ;
[] rr ≠ ww ∨ *pw* = 22 → rr := 1 − rr ; **goto** 54 .
54: **goto** 55 .
55: **goto** 56 .
56: *lc* := cc ; **goto** 57 ;
[] *pw* = 25 → *lc* := 1 − cc ; **goto** 57 .
57: *result* := buf[rr, *lc*] ; **goto** 58 .
58: **goto** 0 .

The nondeterministic choices at 50, 51, 52, 53, 56 reflect the flickering of fw, cc, and ww. We thus eliminate the nonatomicity and the flickering and replace it by nondeterminacy at the point where the variables are inspected. Note that the two processes now inspect each other's program counters. It follows that the skip statements in 22, 25, 27, 54, and 55 cannot be eliminated directly.

We claim specification *HVc2* implements *HVc2a*. This is proved by means of the refinement function *fca* given by

$$
\begin{aligned}
&fca(x) = (\#\\
&\quad \mathtt{ww} := (x.pw = 22\ ?\ 1 - x.\mathtt{wwS} : x.\mathtt{wwS})\ ,\\
&\quad \mathtt{cc} := (x.pw = 25\ ?\ 1 - x.\mathtt{ccS} : x.\mathtt{ccS})\ ,\\
&\quad \mathtt{fw} := (x.\mathtt{fw} \vee x.pw \in \{23, 27\})\ ,\\
&\quad \mathtt{rr} := (x.pr \in \{54, 55\} \vee x.\mathtt{rr} = 2\ ?\ x.lw : x.\mathtt{rr})\ ,\\
&\quad \mathtt{buf} := x.\mathtt{buf}\ ,\ arg := x.arg\ ,\\
&\quad result := x.result\ ,\ lc := x.lc\ ,\\
&\quad pw := x.pw\ ,\ pr := x.pr\\
&\#)\ .
\end{aligned}
$$

Here x is a state of $HVc2$ and the resulting state $fca(x)$ is described as a record in the PVS notation. The correctness proof of this refinement function is based on the invariants $Iq0$ up to $Iq7$ mentioned above. The disjunct $x.\mathtt{rr} = 2$ is needed to define fca everywhere on the state space. It is superfluous on the reachable states because of $Iq5$.

We postpone the question whether this specification implements specification ARW to Section 6 and first resolve the proof obligations for the unsafe variables `buf`.

5.3. Unsafeness

The only unsafe shared variables are the elements of array `buf`. Since they are only read into the private variable *result*, the corresponding proof obligation is that always:

US: $\quad result \neq \bot$.

To prove this, we need to know when $\mathtt{buf}[i, j] = \bot$ holds. Since it is false initially, and since $\bot$ is only written at 21 and 24 when the writer stays in that location, we have the easy invariant

$Jq0$: $\quad \mathtt{buf}[i, j] = \bot \;\Rightarrow\; (pw = 21 \;\wedge\; i = 1 - \mathtt{ww} \;\wedge\; j = \mathtt{cc})$
$\quad\quad \vee \;\; (pw = 24 \;\wedge\; i = \mathtt{ww} \;\wedge\; j = 1 - \mathtt{cc})$.

Actually, in order to prove that this predicate is preserved by the assignments to `buf` in 21 and 24, we need the invariant

$Jq1$: $\quad arg \neq \bot$,

which is obvious since it holds initially and $\bot$ is never chosen in line 20.

Since the reader only reads $\mathtt{buf}[\mathtt{rr}, lc]$, and that only at 57, the invariance of US would follow from the following two derived proof obligations:

$US0$: $\quad pw = 21 \;\wedge\; pr = 57 \;\Rightarrow\; \mathtt{ww} = \mathtt{rr}$,
$US1$: $\quad pw = 24 \;\wedge\; pr = 57 \;\wedge\; \mathtt{ww} = \mathtt{rr} \;\Rightarrow\; \mathtt{cc} = lc$.

These invariants express mutual exclusion: when the writer is writing into a buffer, the reader should not be reading the same buffer.

Recall that an *invariant* is a predicate that holds in all reachable states. A predicate is called *inductive* if it holds initially and is preserved in every step. It is well known that every inductive predicate is an invariant. Below we shall make lists of invariants, and only the conjunctions of these lists will turn out to be inductive, but that is sufficient to prove that all members of the lists are invariants.

One may try and visualize the state space as a high-dimensional grid and the reachable states as forming a rather intractable subset of it. The proof obligations and the invariants are then boxes (hopefully) around the reachable states. Because they are expressed in one or two lines, these boxes consist of a few straight (linear) walls. The proof is the construction of boxes to fend off the forbidden areas. This image is technically useless, but may help the intuition.

There are two approaches to find invariants. The top-down approach starts with the specification, tries to find invariants to prove the specification, and then invariants to support the preservation of the first list of invariants. The bottom-up approach starts with the algorithm, tries to identify potentially useful invariants, and to build upon these, towards the specification. In the top-down approach the invariants tend to be well-motivated but unconvincing. In the bottom-up approach, they tend to be likely, but less to the purpose. The approaches need to complement each other. In the top-down approach, one needs to look at the algorithm to see what is likely to be valid. With the bottom-up approach, one needs to investigate what is needed to imply the specification.

In general, we prefer to follow the top-down approach, both in the design of the proof and in the presentation, because it gives more insight into the logical structure. In the design phase, it is also a way to avoid proving useless invariants, but it entails the risk of introducing invalid invariants.

In the next paragraphs we strongly rely on the proof assistant PVS. We first note that the invariants $Iq2$ and $Iq3$ of algorithm C2 are equally valid in our abstract algorithm, although as predicates on a different state space. We actually need the following slightly stronger versions:

$Iq2a$: $\quad pw \in \{21, 22, 23, 27\} \;\Rightarrow\; \mathtt{fw}$,
$Iq3a$: $\quad pw \in \{24, 25, 26\} \;\Rightarrow\; \neg\mathtt{fw}$.

In order to prove *US0*, it suffices to show that $pw = 21$ implies $\mathtt{rr} = \mathtt{ww}$. This follows from *Iq2a* together with the predicate:

Jq2: $\quad \mathtt{fw} \;\Rightarrow\; \mathtt{rr} = \mathtt{ww} \;\vee\; pw \in \{22, 23\}$.

The validity of *Jq2* is nontrivial. Since ww, fw, and rr are only modified in 21, 23, 26, and 53, and command 23 falsifies fw, predicate *Jq2* is threatened only at 26. It is preserved at 26 because of the new invariant:

Jq3: $\quad pr = 54 \;\Rightarrow\; \mathtt{rr} = \mathtt{ww}$.

Predicate *Jq3* is threatened only at 21 and 53. It is preserved at 21 because of *Iq2a* and *Jq4*, and at 53 because of *Jq5*:

Jq4: $\quad pr = 54 \;\wedge\; \mathtt{fw} \;\Rightarrow\; pw \in \{22, 23\}$,
Jq5: $\quad pr = 53 \;\wedge\; pw = 22 \;\Rightarrow\; \mathtt{rr} = 1 - \mathtt{ww}$.

Predicate *Jq4* is threatened only at 26 and 53. It is preserved at 53 because of *Jq2*, and at 26 because of *Jq3*.

Predicate *Jq5* is preserved at 21, 50, and 52 because of *Iq2a* and *Jq2*. All these predicates also hold initially. This concludes the proof of invariance of *Jq0*.

Predicate *US1* is implied by *Iq3a* together with the new invariant:

Kq0: $\quad pr \in \{52, 57\} \;\wedge\; \mathtt{rr} = \mathtt{ww} \;\wedge\; \neg\mathtt{fw} \;\Rightarrow\; lc = \mathtt{cc} \;\vee\; pw \in \{25, 26\}$.

Predicate *Kq0* is threatened only in 21 and 23. It is preserved at 21 because of *Iq2* and at 23 because of the new invariant:

Kq1: $\quad pr \in \{52, 57\} \;\wedge\; \mathtt{rr} = \mathtt{ww} \;\wedge\; pw \in \{22, 23\} \;\Rightarrow\; lc = \mathtt{cc}$.

Predicate *Kq1* is threatened only in 21. It is preserved because of *Iq2a* and *Jq2*. Again the initializations are trivial. This concludes the proof of the invariant *Kq0*, and therefore of *US0* and *US1*, and hence of *US*.

6. Atomicity

We now turn to the proof that the algorithm implements an atomic read–write system, in the sense that it implements specification *HqRW* of Section 4.2.

6.1. Extending HVc2a towards HVc2b

For this purpose, we extend *HVc2a* with the variables declared for *HqRW*, as ghost variables or history variables. We extend the first and last command of reader and writer with the assignments *first* := masq and masq := max(masq, *sqn*), respectively. Because there is only one writer, the writer can choose *sqnW* := *sqnW* + 1 at the commands where it writes into the buffer, i.e., at 21 and 24.

The reader needs to choose *sqnR* equal to the number *sqnW* that the writer has chosen when writing to the buffer. To make this possible, we introduce a ghost variable:

var tag : **array** *Bit*, *Bit* **of** *Nat* ,

and decide that the writer stores *sqnW* in the field of tag corresponding to the buffer where it is writing. In view of the result of Section 5.3, we can delete the flickering assignments of ⊥ to buf in 21 and 24. The commands 20, 21, 24, and 28 are therefore replaced as follows.

Writer of HVc2b:
20: $\quad$ *firstW* := masq ; **goto** (fw ? 21 : 24) .
21: $\quad$ ww := 1 − ww ; buf[ww, cc] := *arg* ;
$\quad\quad$ *sqnW* ++ ; hist(*sqnW*) := *arg* ;
$\quad\quad$ tag[ww, cc] := *sqnW* ; **goto** 22 .
24: $\quad$ cc := 1 − cc ; buf[ww, cc] := *arg* ; ;
$\quad\quad$ *sqnW* ++ ; hist(*sqnW*) := *arg* ;
$\quad\quad$ tag[ww, cc] := *sqnW* ; **goto** 25 .
28: $\quad$ masq := max(masq, *sqnW*) ; **goto** 0 .

The reader reads its result at 57 and therefore chooses $sqnR$ also at that location. The reader commands 50, 57, and 58 are therefore replaced as follows.

Reader of HVc2b:
50: $firstR := \mathtt{masq}$; **goto** $(\mathtt{fw}\ ?\ 51 : 53)$;
57: $result := \mathtt{buf}[\mathtt{rr}, lc]$; $sqnR := \mathtt{tag}[\mathtt{rr}, lc]$; **goto** 58 ;
58: $\mathtt{masq} := \max(\mathtt{masq}, sqnR)$; **goto** 0 .

The other commands of the abstract specification of Section 5.2 are retained, but are now interpreted on the state space of $HVc2b$.

The initial state is characterized by

$$
\begin{array}{ll}
& (\forall\, i, j : \mathtt{buf}[i, j] = item0 \ \wedge\ \mathtt{tag}[i, j] = 0) \\
\wedge & result = item0 \ \wedge\ arg \neq \bot \\
\wedge & pw = pr = 0 \ \wedge\ (\neg \mathtt{fw} \ \vee\ \mathtt{ww} = \mathtt{rr}) \\
\wedge & \mathtt{masq} = firstW = sqnW = firstR = sqnR = 0 \\
\wedge & \mathtt{hist} = (\lambda\, k : (k = 0\ ?\ item0 : \bot)) \ .
\end{array}
$$

It is straightforward though cumbersome to verify that we have a forward simulation Rab from $HVc2a$ to $HVc2b$ given by

$$
\begin{array}{ll}
& (x, y) \in Rab \quad \equiv \\
& \quad (\forall\, i, j : x.\mathtt{buf}[i, j] = y.\mathtt{buf}[i, j] \\
& \qquad \vee\ (x.pw = 21 \ \wedge\ i \neq x.\mathtt{ww} \ \wedge\ j = x.\mathtt{cc}) \\
& \qquad \vee\ (x.pw = 24 \ \wedge\ i = x.\mathtt{ww} \ \wedge\ j \neq x.\mathtt{cc})) \\
\wedge & \quad x.arg = y.arg \ \wedge\ x.result = y.result \ \wedge\ x.\mathtt{fw} = y.\mathtt{fw} \\
\wedge & \quad x.\mathtt{ww} = y.\mathtt{ww} \ \wedge\ x.\mathtt{rr} = y.\mathtt{rr} \ \wedge\ x.\mathtt{cc} = y.\mathtt{cc} \\
\wedge & \quad x.lc = y.lc \ \wedge\ x.pw = y.pw \ \wedge\ x.pr = y.pr \ .
\end{array}
$$

This forward simulation is a little bit more than an extension with history variables [1], because we also removed the $\bot$ assignments to fields of buf in 21 and 24.

6.2. From HVc2b towards HqRW

In order to show that specification $HVc2b$ implements specification $HqRW$ of Section 4.2, we need to show that the choices of $sqnW$ and $sqnR$ in 21, 24, and 57 satisfy the conditions expressed for $HqRW$, and that the *result* delivered at 58 equals $\mathtt{hist}(sqnR)$.

The steps 21 and 24 of $HVc2b$ satisfy the conditions at 21 in $HqRW$ because of the following assertions:

$Atm0$: $\forall\, i \in \mathbb{N} : sqnW < i \ \Rightarrow\ \mathtt{hist}(i) = \bot$,
$Atm1$: $firstW < sqnW$.

Step 57 of $HVc2b$ satisfies the conditions at 51 of $HqRW$ because of

$Atm2$: $\forall\, i, j : \mathtt{hist}(\mathtt{tag}[i, j]) = \mathtt{buf}[i, j] \neq \bot$,
$Atm3$: $pr = 57 \quad \Rightarrow \quad firstR \leq \mathtt{tag}[\mathtt{rr}, lc]$.

Specification $HVc2b$ also allows a jump from 53 to 58 in which the reader uses its previous result of reading. This jump must also simulate reading as in 51 of $HqRW$. In order to prove this, we postulate the invariants:

$Atm4$: $\mathtt{hist}(sqnR) = result \neq \bot$,
$Atm5$: $pr = 53 \ \wedge\ (\mathtt{ww} = \mathtt{rr} \ \vee\ pw = 22) \quad \Rightarrow \quad firstR \leq sqnR$.

It remains to prove $Atm0$ up to $Atm5$.

We first deal with the easy proof obligations $Atm0$, $Atm1$, $Atm2$, and $Atm4$. It turns out that $Atm0$ itself is inductive because hist is only modified at $sqnW$, and $sqnW$ only increases.

In order to prove $Atm1$, we observe the easy inequalities:

$Lq0$: $\mathit{firstW} \leq \mathtt{masq}$,
$Lq1$: $\mathtt{tag}[p, q] \leq \mathit{sqnW}$,
$Lq2$: $\mathit{sqnR} \leq \mathit{sqnW}$,
$Lq3$: $\mathtt{masq} \leq \mathit{sqnW}$.

Indeed, because masq only increases and *firstW* gets its values from masq, $Lq0$ is invariant. The argument for $Lq1$ is similar. The invariance of $Lq2$ follows from $Lq1$ and the invariance of $Lq3$ follows from $Lq2$. Finally, $Atm1$ is implied by $Lq0$ and $Lq3$. The invariance of $Atm2$ follows from $Lq1$ at the locations 21 and 24, elsewhere it is obvious. The invariance of $Atm4$ follows from $Lq2$ at 21 and 24, and from $Atm2$ at 57. This leaves $Atm3$ and $Atm\ 5$ as proof obligations, but these ones are difficult.

6.3. Upper bounds for firstR

Predicate $Atm3$ expresses that the item read by the reader is not too old. As a first approximation we note the easy invariants

$Lq4$: $\mathit{firstR} \leq \mathtt{masq}$,
$Lq5$: $\mathit{sqnW} = \mathtt{tag}[\mathtt{ww}, \mathtt{cc}]$.

We also note that the invariants $Iq2a$, $Iq3a$, $Jq1$, ..., $Jq5$, $Kq0$, $Kq1$ of $HVc2a$ are easily extended literally to specification $HVc2b$. Because they are defined on a different state space, we give these extended invariants the names $Iq2b$, $Iq3b$, $Jq1b$, ..., $Kq1b$.

The proof of invariance of $Atm3$ is so delicate that one may prefer to skip it. It was designed in the top-down fashion. For each invariant, we proceed as follows. We ask PVS to prove its invariance, we note the locations where it cannot prove this, as well as the failing conditions. The failing conditions are then used to postulate new invariants. Creativity is needed to generalize and unify the invariants. The litany that follows is a careful report of the result of theorem proving, but the reader should not try to reproduce all these verifications manually. This part of the proof feels to us like climbing a winding staircase described by the four fields of tag.

Predicate $Atm3$ is threatened only at the locations 21, 24, 52, and 56. It is preserved at 21 and 24 because of $Lq3$ and $Lq4$. It is preserved at 52 and 56 because of $Iq2b$ and the new invariants:

$Mq0$: $\mathtt{fw} \wedge pr = 52 \Rightarrow \mathit{firstR} \leq \mathtt{tag}[\mathtt{rr}, lc]$,
$Mq1$: $pr \in \{54 \ldots 56\} \Rightarrow \mathit{firstR} \leq \mathtt{tag}[\mathtt{rr}, \mathtt{cc}]$,
$Mq2$: $pw \in \{21, 25 \ldots 27\} \wedge pr \in \{54 \ldots 56\} \Rightarrow \mathit{firstR} \leq \mathtt{tag}[\mathtt{rr}, 1 - \mathtt{cc}]$.

Predicate $Mq0$ is threatened at 21, 24, 26, and 51. It is preserved at 21 and 24 because of $Lq3$ and $Lq4$. It is preserved at 26 and 51 because of $Iq3b$ and the new predicates

$Mq3$: $pw \in \{25, 26\} \wedge pr = 52 \wedge \mathtt{ww} = \mathtt{rr} \Rightarrow \mathit{firstR} \leq \mathtt{tag}[\mathtt{rr}, lc]$,
$Mq4$: $\mathtt{fw} \wedge pr = 51 \Rightarrow \mathit{firstR} \leq \mathtt{tag}[\mathtt{rr}, \mathtt{cc}]$.

Predicate $Mq1$ is threatened at 21, 24 and 53. It is preserved at 21 because of $Lq3$ and $Lq4$. It is preserved at 24 and 53 because of $Lq3$, $Lq4$, $Lq5$, $Jq5b$, and the new predicate

$Mq5$: $pr \in \{54 \ldots 56\} \Rightarrow \mathtt{ww} = \mathtt{rr} \vee \mathit{firstR} \leq \mathtt{tag}[\mathtt{rr}, 1 - \mathtt{cc}]$.

Predicate $Mq2$ is threatened at 20, 24 and 53. It is preserved at these locations because of $Iq2b$, $Jq2b$, $Mq1$, and the new predicates

$Mq6$: $\mathtt{fw} \wedge pw \notin \{22, 23\} \wedge pr \in \{54 \ldots 56\} \Rightarrow \mathit{firstR} \leq \mathtt{tag}[\mathtt{rr}, 1 - \mathtt{cc}]$,
$Mq7$: $pw \in \{25 \ldots 27\} \wedge pr \in \{51 \ldots 53\}$
$\Rightarrow \mathtt{ww} = \mathtt{rr} \vee \mathit{firstR} \leq \mathtt{tag}[\mathtt{ww}, 1 - \mathtt{cc}]$.

Predicate $Mq3$ is threatened at 24 and 51. It is preserved because of $Lq3$, $Lq4$, $Lq5$, and the new predicate

$Mq8$: $pw = 25 \wedge pr = 51 \wedge \mathtt{ww} = \mathtt{rr} \Rightarrow \mathit{firstR} \leq \mathtt{tag}[\mathtt{rr}, 1 - \mathtt{cc}]$.

Predicate $Mq4$ is threatened at 21, 24, 26, and 50. It is preserved at 21, 24, and 26 because of $Lq3$, $Lq4$, $Lq5$, and $Iq3b$. It is preserved at 50 because of $Jq2b$, $Lq3$, $Lq5$, and the new predicate

$Nq0$: $\quad pw \in \{22, 23\} \;\Rightarrow\; \mathtt{masq} \leq \mathtt{tag}[\mathtt{rr}, \mathtt{cc}]$.

Predicate $Mq5$ is threatened at 21, 24 and 53. It is preserved because of $Mq1$, $Mq2$, and $Jq5b$.

Predicate $Mq6$ is threatened at 24, 26, and 53. It is preserved because of $Iq3b$, $Jq2b$, $Jq3b$ and $Mq2$.

Predicate $Mq7$ is threatened at 24 and 50. It is preserved because of $Lq3$, $Lq4$, $Lq5$, $Iq2b$, $Jq2b$, and the new predicate

$Pq0$: $\quad pw \in \{25, 26\} \;\wedge\; pr \leq 50 \;\Rightarrow\; \mathtt{ww} = \mathtt{rr} \;\vee\; \mathtt{masq} \leq \mathtt{tag}[\mathtt{ww}, 1 - \mathtt{cc}]$.

Predicate $Mq8$ is threatened at 24 and 50. It is preserved because of $Lq3$, $Lq4$, $Lq5$, and $Iq3b$. This concludes the list of proofs of invariants about upper bounds of *firstR*.

6.4. The remainder of the proof of A tm3

It remains to prove the two invariants $Nq0$, $Pq0$ about masq. These give rise to upper bounds for *sqnR* and tag.

As before, we frequently apply the invariants $Iq2b$, $Iq3b$, $Jq2b$, and the invariants of the family $Lq*$. Let us call them the basic invariants. We henceforth focus on the more uncommon invariants, and only mention application of the basic invariants when there are no other invariants involved.

Predicate $Nq0$ is threatened at 21, 53, and 58. It is preserved because of $Jq5b$ and the new predicate

$Nq1$: $\quad pw \in \{22, 23\} \;\wedge\; pr = 58 \;\Rightarrow\; sqnR \leq \mathtt{tag}[\mathtt{rr}, \mathtt{cc}]$.

Predicate $Nq1$ is threatened at 21, 53, and 57. It is preserved because of the new predicates

$Nq2$: $\quad pw = 22 \;\wedge\; pr = 53 \;\Rightarrow\; sqnR \leq \mathtt{tag}[\mathtt{rr}, \mathtt{cc}]$,
$Nq3$: $\quad pw \in \{22, 23\} \;\wedge\; pr \in \{52, 57\} \;\Rightarrow\; \mathtt{tag}[\mathtt{rr}, lc] \leq \mathtt{tag}[\mathtt{rr}, \mathtt{cc}]$.

Predicate $Nq2$ is threatened at 21, 50, and 52, and $Nq3$ is threatened at 21. Both are preserved because of basic invariants. It is likely that we cannot extend $Nq2$ with $pw = 23$.

Predicate $Pq0$ is threatened at 24 and 58. It is preserved because of the new predicates

$Pq1$: $\quad pw \in \{25, 26\} \;\Rightarrow\; \mathtt{ww} = \mathtt{rr} \;\vee\; \mathtt{masq} \leq \mathtt{tag}[\mathtt{ww}, 1 - \mathtt{cc}]$,
$Pq2$: $\quad pw \in \{25, 26\} \;\wedge\; pr = 58 \;\Rightarrow\; \mathtt{ww} = \mathtt{rr} \;\vee\; sqnR \leq \mathtt{tag}[\mathtt{ww}, 1 - \mathtt{cc}]$.

Predicate $Pq1$ is threatened at 24 and 58. It is preserved because of $Pq2$. Predicate $Pq2$ is threatened at 24 and 57. It is preserved because of the new predicate

$Pq3$: $\quad pw \in \{25, 26\} \;\wedge\; pr = 57$
$\quad\quad \Rightarrow\; \mathtt{ww} = \mathtt{rr} \;\vee\; \mathtt{tag}[\mathtt{rr}, lc] \leq \mathtt{tag}[\mathtt{ww}, 1 - \mathtt{cc}]$.

Predicate $Pq3$ is threatened at 24, 52, and 56. It is preserved because of the new predicates

$Pq4$: $\quad pw \in \{25, 26\} \;\wedge\; pr \in \{54 \ldots 56\}$
$\quad\quad \Rightarrow\; \mathtt{ww} = \mathtt{rr} \;\vee\; \mathtt{tag}[\mathtt{rr}, \mathtt{cc}] \leq \mathtt{tag}[\mathtt{ww}, 1 - \mathtt{cc}]$,
$Pq5$: $\quad pw = 25 \;\wedge\; pr \in \{54 \ldots 56\} \;\Rightarrow\; \mathtt{tag}[\mathtt{rr}, 1 - \mathtt{cc}] \leq \mathtt{tag}[\mathtt{ww}, 1 - \mathtt{cc}]$.

Predicates $Pq4$ and $Pq5$ are both threatened only at 24. They are both preserved because of $Lq1$ and $Lq5$. This concludes the proof of $A\,tm3$.

6.5. The proof of A tm5

Predicate $A\,tm5$ is threatened only at 21, 50 and 52. It is preserved because of the new predicate

$Qq0$: $\quad (\neg \mathtt{fw} \;\vee\; pw \in \{22 \ldots 27\}) \;\wedge\; pr \notin \{54 \ldots 57\} \;\wedge\; \mathtt{ww} = \mathtt{rr}$
$\quad\quad \Rightarrow\; \mathtt{masq} \leq sqnR$.

Predicate *Qq0* is threatened at 21, 28, and 57. It is preserved because of *Kq0b*, *Kq1b*, and the new predicates

$$Qq1: \quad \neg \mathtt{fw} \ \wedge \ pw = 28 \ \wedge \ pr \notin \{54 \ldots 57\} \ \wedge \ \mathtt{ww} = \mathtt{rr} \ \Rightarrow \ sqnW = sqnR \ ,$$
$$Qq2: \quad pw \in \{25 \ldots 27\} \ \wedge \ pr \in \{52, 57\} \ \wedge \ \mathtt{ww} = \mathtt{rr} \ \Rightarrow \ \mathtt{masq} \leq \mathtt{tag}[\mathtt{ww}, lc] \ .$$

Predicate *Qq1* is threatened at 23, 27, and 57. It is preserved because of *Kq0b* and the new predicate

$$Qq3: \quad pw \in \{22, 23\} \ \wedge \ pr \notin \{54 \ldots 57\} \ \wedge \ \mathtt{ww} = \mathtt{rr} \ \Rightarrow \ sqnW = sqnR \ .$$

Predicate *Qq3* is threatened at 21 and 57. It is preserved because of *Kq1b* and some basic invariants.

Predicate *Qq2* is threatened at 24, 51, and 56. It is preserved because of *Kq0b* and the new predicate

$$Qq4: \quad pw = 25 \ \wedge \ pr \in \{51, 54 \ldots 56\} \ \wedge \ \mathtt{ww} = \mathtt{rr}$$
$$\Rightarrow \ \mathtt{masq} \leq \mathtt{tag}[\mathtt{ww}, 1 - \mathtt{cc}] \ .$$

Predicate *Qq4* is threatened at 24, 50, and 53. It is preserved because of *Pq1*. This concludes the proof of *Atm5*.

6.6. Refinement towards HqRW

We prove that specification *HVc2b* implements *HqRW* by means of a refinement function *fbh* from the state space *XB* of *HVc2b* to the state space *XH* of *HqRW*. Function *fbh* preserves the variables that occur in both spaces: *arg*, *result*, hist, masq, *firstW*, *sqnW*, *firstR*, *sqnR*. It removes the other implementation variables, and it compresses the locations. Therefore, for $x \in XB$, the program counters of $fbh(x)$ satisfy:

$$fbh(x).pw = (x.pw \leq 20 \ ? \ x.pw \ : \ x.pw \in \{21, 24\} \ ? \ 21 : 22) \ .$$
$$fbh(x).pr = (x.pr \leq 50 \ ? \ x.pr \ : \ x.pr \leq 57 \ ? \ 51 : 52) \ .$$

Using the atomicity predicates *Atm0*, ..., *Atm5*, we proved that *fbh* indeed is a refinement function from *HVc2b* to *HqRW*.

Because simulation relations are transitive, we thus have a simulation from *HVc2* to *HqRW*. This simulation preserves the values of *arg* and *result* and the conditions $pw = 0$ and $pr = 0$. It therefore preserves observational equivalence. This concludes the proof of correctness of algorithm C2.

7. Two safe bits instead of the regular variable

It is possible to replace the three-valued regular variable rr of algorithm C2 by two safe bits, say ra and rb, initially equal. The idea is that ra and rb represent the value of rr when they are equal, and that rr = 2 when ra and rb differ. In the style of Section 3, the reader's lines concerning rr are changed into

54: (flickering) ra := *lw* ;
55: (flickering) rb := *lw* ;
57: *result* := buf[ra, *lc*] .

The writer's conditional statement based on the test of rr is replaced by

26: **if** ra = ww **then**
30: **if** rb = ww **then**
27: (flickering) fw := *true* **end end** .

In the specification corresponding to *HVc2*, the writer needs an additional location (30) because its test of rr is replaced by two atomic tests.

This alternative specification *HVac2* also allows a refinement function towards *HVc2a*. This function maps a state *x* of *HVac2* to the abstract state

$$
\begin{array}{l}
(\#\ \mathtt{ww} := (x.pw = 22\ ?\ 1 - x.\mathtt{wwS} : x.\mathtt{wwS})\ , \\
\quad \mathtt{cc} := (x.pw = 25\ ?\ 1 - x.\mathtt{ccS} : x.\mathtt{ccS})\ , \\
\quad \mathtt{fw} := (x.\mathtt{fw} \vee x.pw \in \{23, 27\})\ , \\
\quad \mathtt{rr} := (x.pr = 54\ ?\ x.lw : x.\mathtt{ra})\ , \\
\quad pw := (x.pw = 30\ ?\ 26 : x.pw)\ , \\
\quad \mathtt{buf} := x.\mathtt{buf}\ ,\ arg := x.arg\ , \\
\quad result := x.result\ ,\ lc := x.lc\ ,\ pr := x.pr \qquad \#)\ .
\end{array}
$$

In order to prove that this is indeed a refinement function, we need a number of invariants, namely $Iq0$, $Iq1$, $Iq2$ as in Section 5.1, and the following four invariants

$$
\begin{array}{l}
pr \notin \{54, 55\} \quad \Rightarrow \quad \mathtt{ra} = \mathtt{rb}\ , \\
pr = 54 \quad \Rightarrow \quad \mathtt{rb} \neq lw\ , \\
pr = 55 \quad \Rightarrow \quad \mathtt{ra} = lw\ , \\
pw = 30 \quad \Rightarrow \quad \mathtt{ww} = \mathtt{ra} \vee pr = 54\ .
\end{array}
$$

In order to prove the last one, we need to introduce several other invariants which are analogues of the invariants $Jq*$ in Section 5.3. The details can be found in the mechanical proof.

8. Conclusions

It is satisfactory that the algorithm is correct and that we were able to verify this. The structure of the proof with one forward simulation between two refinement functions is also quite satisfactory. The part of the proof in the sections 6.3 up to 6.5 is less satisfactory, in that we had to invent several unintuitive invariants.

Haldar and Vidyasankar verified the algorithm by operational arguments using Lamport's precedence relation between events, cf. [7]. Since we have not seen this proof, we cannot make a comparison. We assume, however, that this operational proof should be at least as delicate as the refinement proof given here. In general, we greatly prefer assertional proofs over operational ones, but the present proof is a weak example. The challenge for assertional methods is of course to provide a more elegant assertional proof, or perhaps even a derivation of the algorithm.

It seems likely that the correctness of algorithm C2 can be confirmed by model checking. This would depend on the fact that there are only one writer and only one reader, so that by some general argument the number of elements of the type *Item* could be reduced to 2. We must leave this to future research.

Acknowledgements

Constructive comments and suggestions of J. F. Groote, S. Haldar, L. Lamport, and one anonymous referee are gratefully acknowledged. In particular, by model checking the algorithm of Section 3 of a previous draft, Groote discovered a severe mistake, discrepancy with the formal model of Section 5, which has been amended here.

References

[1] M. Abadi, L. Lamport, The existence of refinement mappings, Theoret. Comput. Sci. 82 (1991) 253–284.
[2] B. Bloom, Constructing two-writer atomic registers, IEEE Trans. Comput. 37 (1988) 1506–1514.
[3] S. Haldar, K. Subramanian, Space-optimum conflict-free construction of 1-writer 1-reader multivalued atomic variable, in: Proceedings of the 8th International Workshop on Distributed Algorithms, in: LNCS, vol. 857, Springer-Verlag, 1994, pp. 116–129.
[4] W.H. Hesselink, An assertional criterion for atomicity, Acta Inf. 38 (2002) 343–366.
[5] W.H. Hesselink, An assertional proof for a construction of an atomic variable, Form. Asp. Comput. 16 (2004) 387–393.
[6] W.H. Hesselink, A criterion for atomicity revisited, Acta Inf. 44 (2007) 123–151.
[7] L. Lamport, On interprocess communication. Parts I and II, Distrib. Comput. 1 (1986) 77–101.
[8] L. Lamport, The temporal logic of actions, ACM Trans. Program. Lang. Syst. 16 (1994) 872–923.
[9] N.A. Lynch, Distributed Algorithms, Morgan Kaufman, San Francisco, 1996.
[10] S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert, PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference, 2001, http://pvs.csl.sri.com.
[11] P.M.B. Vitányi, B. Awerbuch, Atomic shared register access by asynchronous hardware, in: 27th Annual Symposium on Foundations of Computer Science, IEEE, Los Alamitos, Calif., 1986, pp. 233–243. Corrigendum in 28th Annual Symposium on Foundations of Computer Science, Los Angeles, 1987, p. 487.